

Hash Tables and Hash Function

Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Introduction to Algorithms by Cormen et. al

Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✓	<code>compareTo()</code>

Q: Can we do better?

A: Yes, but with different access to the data.

Hash table: Introduction

- **Hash Table:** is a data structure used to implement Dynamic Sets/Symbol Table.
- Operations: $\text{FIND}(x)$ / $\text{SEARCH}(x)$, $\text{INSERT}(x)$, $\text{DELETE}(x)$
- We want $O(1)$, at least on average, for each operation!
- Typically used when the number of keys stored is far less than all the possible keys.

Array/DAT Vs. Hash table

- Directly addressing into an ordinary array enables us to search, insert and delete arbitrary elements in an array in $O(1)$ time
 - Impractical/impossible if the universe $|U|$ is large, and results in memory wastage if the set of keys actually stored $|K| \ll |U|$.

1	53	257	10,000	20,004
1	53	257	10,000	20,004

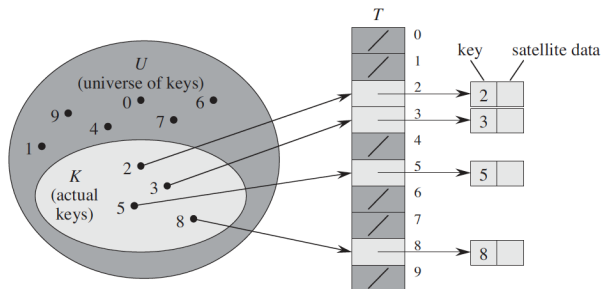
- Can we do better/ Yes, use **hash tables**.
 - Hash table typically uses an array of size proportional to the number of keys actually stored.
 - Hash table – Instead of using the key as an array index directly, the array index is computed from the key using a **hash function**.

$h(10,000)$ $h(1)$ $h(257)$ $h(53)$ $h(20,004)$

10,000	1	257	53	20,004
--------	---	-----	----	--------

When to use a hash table?

- Universe of keys $U = \{0, 1, \dots, n\}$ is very large, and actually key set $K = \{0, 1, \dots, m-1\}$ is small.
- Simply use an array/direct address table (DAT) $T[0..m-1]$.
- Every slot corresponds to a **unique key** in the universe U .
- If a key k absent in actual key set $K \Rightarrow T[k] = \text{NIL}$ (dark grey slots).



Hash table I

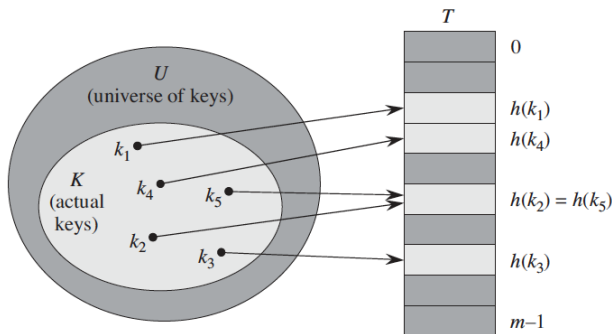
- With direct addressing, an element with key k is stored in slot k .
- With hashing, this element is stored in slot $h(k)$, where h is the **hash function**. h maps the universe U of keys into the slots of a **hash table** $T[0..m-1]$

$$h : U \rightarrow \{0, 1, \dots, m-1\}, \text{ where } m \ll |U|$$

- We say that an element with key k **hashes** to slot $h(k)$; we also say that $h(k)$ is the **hash value** of key k .
- The hash function reduces array size from $|U|$ to m = actually number of keys stored.

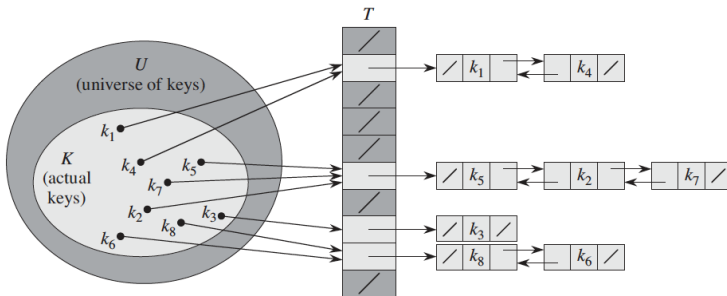
Hash table: Problem

- Problem: Since $m \ll |U|$, two keys may hash to the same slot; that is, for $k_1, k_2 \neq k_1 \in U, h(k_1) = h(k_2)$. This is called **collision**.

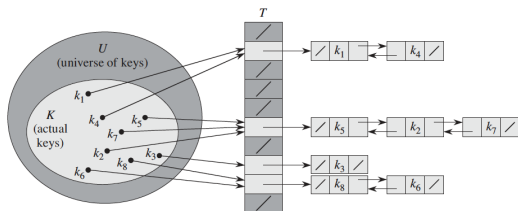


Hash table: Collision resolution by chaining I

- In **chaining**, we place all the elements that hash to the same slot into the same linked list.
- Slot j contains a pointer to the head of the list of all stored elements that hash to j ; if there are no such elements, slot j contains NIL.
- For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.



Hash table: Collision resolution by chaining II



- **INSERT(k)**: insert k at the head of list $T(h(k))$ – time complexity $O(1)$ (Symbol table has no duplicates – assume k is not in list).
- **DELETE(k)**: delete k from the list $T(h(k))$ – time complexity, worst case $O(m)$, where $m = |T|$
- **FIND(k)**: search for an element k in the list $T(h(k))$ – time complexity, worst case $O(m)$

It is much better on average for 'good' hash functions h !

Good Hash Function - I

- The probability that $h(k_1) = h(k_2)$ for $k_1 \neq k_2$ is “small”.
- A good hash function satisfies (approximately) the assumption of –
Simple Uniform Hashing: Each element of U is equally likely to hash to any of the m slots, independently of where any other element of U has hashed to.
 - Suppose $U = \{0, 1, \dots, n-1\}$, $|T| = m$ and all elements of U are uniformly distributed. The function:

$$h(k) = \left\lfloor \frac{k}{n} m \right\rfloor$$

satisfies (approximately) the condition of simple uniform hashing.

- For example, consider $n = 100, m = 10$. Then, keys in the range $0 - 9$ get hashed to slot 0, keys in the range $10 - 19$ get hashed to slot 1 and so on.
- Most hash functions assume that the universe of keys is the same of natural numbers $U = \{0, 1, 2, 3, \dots\}$.

Good Hash Function - II

- **Problem:** One rarely knows the probability distribution according to which elements of U are drawn.
- In practice, we can often employ heuristic techniques to create a hash function that performs well.
- Qualitative information about the distribution of keys may be useful in this design process. For example, consider a compiler's symbol table, in which the keys are character strings representing identifiers in a program. Closely related symbols, such as `pt` and `pts`, often occur in the same program. A good function would minimize such variants to hash to the same slot.
- Note that some applications of hash functions might require stronger properties than that are provided by simple uniform hashing. For instance, if you require that the keys that are close, DO NOT hash to the same slot (as in case of linear probing).

'Good' Hash Functions - Division method I

In the **Division method**, we map a key k into one of m slots by taking the remainder of k divided by m ; that is, the hash function is

$$h(k) = k \bmod m$$

For example, if the hash table (T) has size $m = 12$, and the key is $k = 100$, then $h(k) = h(100) = 4$.

- Hashing by division is quite fast – requires only a single division operation.
- Not all choices of m are good, m should satisfy $m \neq 2^p$, a prime number not too close to 2^p is a good choice.
- If $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k . Unless we know that all low-order p -bit patterns are equally likely, we are better off designing the hash function to depend on all the bits of the key.

'Good' Hash Functions - Division method II

In the **division method**, we map a key k into one of m slots by taking the remainder of k divided by m ; that is, the hash function is

$$h(k) = k \bmod m$$

A prime number not too close to 2^p is a good choice.

- For example, suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly $n = 2000$ character strings, where a character has 8 bits. We don't mind examining an average of 3 elements in an unsuccessful search, and so we allocate a hash table of size $m = 701$. We could choose $m = 701$ because it is a prime near $2000/3 = 666.66$ but not near any power of 2.
- Treating each key k as an integer, our hash function would be

$$h(k) = k \bmod 701$$

'Good' Hash Functions - Multiplication method II

The **multiplication method** for creating hash functions operates in two steps.

- We multiply the key k by a constant A in the range $0 < A < 1$, and extract the fraction part $kA \bmod 1$, which is equal to $(kA \bmod 1)$.
- Then, we multiply $kA \bmod 1$ by m and take the floor.
- Hence the hash function is:

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

- Example: $m = 4, k = 105, 106, A = 0.5$, then

$$\begin{aligned} h(k) &= \lfloor m(kA \bmod 1) \rfloor \\ &= \lfloor 4(105 \times 0.5 \bmod 1) \rfloor \\ &= \lfloor 4(52.5 \bmod 1) \rfloor \\ &= \lfloor 4 \times 0.5 \rfloor \\ &= 2 \end{aligned}$$

$$\begin{aligned} h(k) &= \lfloor m(kA \bmod 1) \rfloor \\ &= \lfloor 4(106 \times 0.5 \bmod 1) \rfloor \\ &= \lfloor 4(53 \bmod 1) \rfloor \\ &= \lfloor 4 \times 0 \rfloor \\ &= 0 \end{aligned}$$

'Good' Hash Functions - Multiplication method II

- An advantage of this method is that the value of m is not critical.
- We typically choose it to be a power of 2 ($m = 2^p$ for some integer p), so an implementation of $h(k)$ is easy.
- This method works with any A , it works better with some values than with others.
- Knuth suggests that $A \approx (\sqrt{5} - 1)/2 = 0.6180339887$ works well.

Open Addressing

In **open addressing**, all elements are stored in the hash table itself (no chaining!).

We will study three open addressing schemes.

- 1 Linear Probing
- 2 Quadratic Probing
- 3 Double hashing

Linear Probing

Let $h' : U \mapsto \{0, 1, \dots, m - 1\}$ be some given 'good' hash function (called auxiliary hash function). Then **Linear Probing** uses the hash function:

$$h(k, i) = (h'(k) + i) \mod m, \text{ for } i = 0, 1, \dots, m - 1$$

i is sometimes called the **iterative index**.

Given key k , the first slot probed is $T[h'(k)]$, if it is occupied we probe $T[h'(k) + 1]$, if occupied we probe $T[h'(k) + 2]$, and so on.

Linear Probing Example

Example: Let $h'(k) = k \bmod 17$ and $h(k, i) = (h'(k) + i) \bmod m$. Suppose we want to store keys 100, 67, 66.

For $k = 100$, we have $h(k, 0) = ((100 \bmod 17) + 0) \bmod 13$,
 $h(k, 0) = (15 + 0) \bmod 13 = 2$.

For $k = 67$, we have $h(k, 0) = ((67 \bmod 17) + 0) \bmod 13 = (16 + 0) \bmod 13 = 3$.

For $k = 66$, we have $h(k, 0) = ((66 \bmod 17) + 0) \bmod 13 = (15 + 0) \bmod 13 = 2$, but since its filled we use the next $i = 1$ value.

$h(k, 1) = ((66 \bmod 17) + 1) \bmod 13 = (15 + 1) \bmod 13 = 3$, but since its filled we use the next $i = 2$ value.

$h(k, 2) = ((66 \bmod 17) + 2) \bmod 13 = (15 + 2) \bmod 13 = 4$. Therefore 66 goes in slot 4.

		100	67	66							
--	--	-----	----	----	--	--	--	--	--	--	--

Linear Probing

- Open addressing is easy to implement, but it suffers from **primary clustering** problem – Long runs of occupied slots build up, increasing the average search time.
- Long runs of occupied slots tend to get longer and avg. search time rises.

Quadratic Probing

Quadratic Probing uses a hash function of the form

$$h(k, i) = (h_0(k) + c_1i + c_2i^2) \mod m,$$

where h_0 is an auxiliary 'good' hash function, and $c_1 \neq 0$ and $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \dots, m - 1$.

The initial probe is $T[h_0(k)]$, later positions probed are at an offset by amounts that depend in a quadratic manner on the number i .

Quadratic Probing Example

Example: Let $h'(k) = k \bmod 17$, $c_1 = 2$, $c_2 = 3$, and $h(k, i) = (h'(k) + i + i^2) \bmod 13$. Suppose we want to store keys 100, 67, 66.

For $k = 100$, we have $h(k, 0) = ((100 \bmod 17) + 0 + 0) \bmod 13$, $h(k, 0) = 15 \bmod 13 = 2$.

For $k = 67$, we have $h(k, 0) = ((67 \bmod 17) + 0 + 0) \bmod 13 = 16 \bmod 13 = 3$.

For $k = 66$, we have $h(k, 0) = ((66 \bmod 17) + 0 + 0) \bmod 13 = 15 \bmod 13 = 2$, but since its filled we use the next $i = 1$ value.

$h(k, 1) = ((66 \bmod 17) + 2 + 3) \bmod 13 = (15 + 5) \bmod 13 = 7$. Therefore 66 goes in slot 7.

		100	67			66					
--	--	-----	----	--	--	----	--	--	--	--	--

Quadratic Probing

- 1 Quadratic Probing works much better than linear probing, and clustering can be avoided.
- 2 However, if two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$.
- 3 This property leads to a milder form of clustering, called **secondary clustering**.

Double Hashing

Double hashing is one of the best method available for open addressing.

Double hashing uses a hash function of the form

$$h(k, i) = (h_1(k) + ih_2(k)) \mod m,$$

where h_1 and h_2 are auxiliary ‘good’ hash functions, and i is an iterative index.

- The initial probe is into the table T is at $T[h_1(k)]$ successive probe positions (for $i = 1, 2, \dots$) are at an offset from previous positions by the amount $h_2(k)$ modulo m .
- Thus, unlike the case of linear or quadratic probing, the probe sequence here depends in two ways upon the key k , since the initial probe position, the offset, or both, may vary.
- The performance of double hashing is very close to the performance of the “ideal” scheme of uniform hashing.

Double hashing Example

Example: Let $h_1(k) = k \bmod 17$, $h_2(k) = k \bmod 7$, and $h(k, i) = (h_1(k) + ih_2(k)) \bmod 13$. Suppose we want to store keys 100, 67, 66.

For $k = 100$, we have $h(k, 0) = ((100 \bmod 17) + 0) \bmod 13$, $h(k, 0) = 15 \bmod 13 = 2$.

For $k = 67$, we have $h(k, 0) = ((67 \bmod 17) + 0) \bmod 13 = 16 \bmod 13 = 3$.

For $k = 66$, we have $h(k, 0) = ((66 \bmod 17) + 0) \bmod 13 = 15 \bmod 13 = 2$, but since its filled we use the next $i = 1$ value.

$h(k, 1) = ((66 \bmod 17) + (66 \bmod 7)) \bmod 13 = (15 + 3) \bmod 13 = 5$.
Therefore 66 goes in slot 5.

		100	67		66						
--	--	-----	----	--	----	--	--	--	--	--	--

Double Hashing

- Double hashing works better when $h_2(k)$ is relative prime to the hash-table size m for the entire hash table to be searched.
- One way is to let m be prime and to design h_2 so that it always returns a positive integer less than m .
- For example, we could choose m prime and let:
$$h_1(k) = k \bmod m$$
$$h_2(k) = 1 + (k \bmod m')$$
where m' is chosen to be slightly less than m (say, $m - 1$).

Double hashing Example

Example: Let $h_1(k) = k \bmod 17$, $h_2(k) = 1 + (k \bmod 12)$, and $h(k, i) = (h_1(k) + ih_2(k)) \bmod 13$. Suppose we want to store keys 100, 67, 66.

For $k = 100$, we have $h(k, 0) = ((100 \bmod 17) + 0) \bmod 13$, $h(k, 0) = 15 \bmod 13 = 2$.

For $k = 67$, we have $h(k, 0) = ((67 \bmod 17) + 0) \bmod 13 = 16 \bmod 13 = 3$.

For $k = 66$, we have $h(k, 0) = ((66 \bmod 17) + 0) \bmod 13 = 15 \bmod 13 = 2$, but since its filled we use the next $i = 1$ value.

$h(k, 1) = ((66 \bmod 17) + 1 + (66 \bmod 12)) \bmod 13 = (15 + 1 + 6) \bmod 13 = 9$. Therefore 66 goes in slot 5.

		100	67					66			
--	--	-----	----	--	--	--	--	----	--	--	--